

design ideas

Edited by Bill Travis and Anne Watson Swager

Simple BER meter is easy to build

Luis Miguel Brugarolas, Sire Sistemas y Redes Telmáticas, Tres Cantos, Madrid, Spain

A BIT-ERROR-RATE (BER) tester is a basic tool for digital-communications measurements. Although many commercial BER testers are available, you can easily design and build an inexpensive version. The scheme in **Figure 1** has performance similar to that of a commercial tester but requires you to perform a manual calculation based on displayed data. The tester displays received bits and received erroneous bits, and you must calculate the BER data using a handheld calculator, for example.

You can build the tester in **Figure 1** from a piece of programmable logic, such as an FPGA or a CPLD, and two counter modules. You can buy the counter modules in kit form or in built form from numerous suppliers. The counters are available in LCD or LED-display formats with four or more digits. The counter modules must have overflow indicators, and they must allow pulse widths that are as narrow as half the data-clock period.

Figure 2 shows the core of the error detector. This detector uses the same pseudorandom-bit-sequence (PRBS)

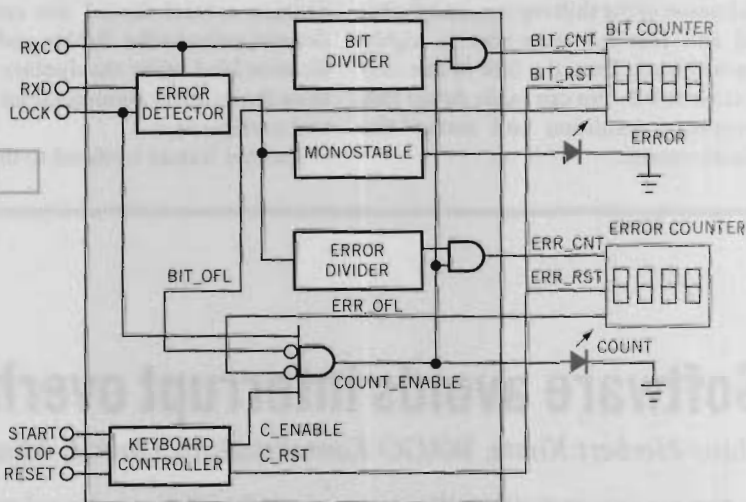


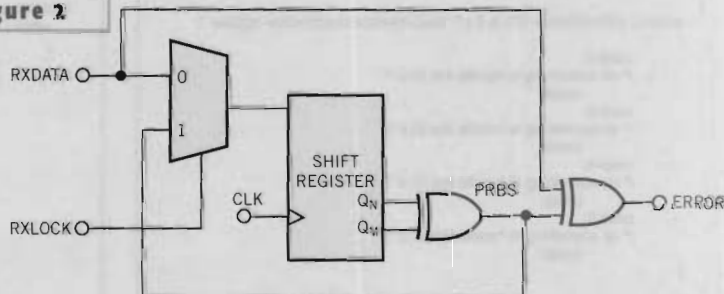
Figure 1

With some programmable logic and two counter modules, you can design and build a simple bit-error-rate tester.

generator as the transmitter does but adds a trick. When the demodulator under test is not in lock, the shift register loads the receive data, and no error count takes place. In every demodulator, except for special burst-mode units, the BER de-

creases to some nominal rate before you declare the system "locked." Thus, the shift register is self-synchronized to the incoming sequence with great probability. When the demodulator is in lock, the Lock signal switches the multiplexer out-

Figure 2



In the error-detector core, the Lock signal controls whether the shift register loads with received data or the locally generated pseudorandom bit sequence (PRBS).

Simple BER meter is easy to build	115
Software avoids interrupt overhead	116
Delay line eases Spice dead-time generation	118
Comparison macro for PIC processors	122
Bridging enhances filter close-in selectivity	122
Notch filter is insensitive to component tolerances	126

put so that locally generated data, which should be the same data as the transmitted data, shifts into the register while the measurement is running. Any divergence between the received data and the locally generated data constitutes a bit error. As long as the counter counts pulses, the error signal must combine with the clock in an RZ format so that the tester does not count two consecutive errors as only one.

An error that occurs just before the demodulator is in lock causes incorrect initialization of the shift register, and the local and received sequences is highly uncorrelated. Thus, the BER in this case is close to 0.5. You can easily detect this erroneous condition and restart the measurement.

The architecture in **Figure 1** allows you to divide the number of errors and bits in the error-divider and bit-divider blocks. You can divide errors by 1, 10, 100 and 1000 and divide bits by 10^4 , 10^5 , 10^6 , and 10^7 . This division feature allows the tester to measure a range of BERs from poor ones for which the tester must divide the error count to a low value to situations that require bit division or that entail long measurement periods. Two switches for each block can control the division rate in a simple way. To avoid mistakes, division control should also control a decimal point in the display, and an indication label under the displays should show the multiplication factor for the actual configuration.

Another feature is related to the over-

flow. When either counter unit overflows, the scheme in **Figure 1** immediately stops the error count using the BIT_OFL and ERR_OFL flags so that the tester does not display erroneous data when taking unattended measurements or when taking measurements for long periods. When active, the BIT_OFL and ERR_OFL flags turn off the COUNT_ENABLE signal.

The Start, Stop, and Reset keys control the unit. They drive a finite state machine, which produces the variables C_ENABLE and C_RESET. The first variable controls the bit and error count, and the second controls the counter reset. (DI #2488)

TO VOTE FOR THIS DESIGN,
CIRCLE NO. 406

Software avoids interrupt overhead

Hans-Herbert Kirste, WAGO Kontakttechnik GmbH, Minden, Germany

YOU CAN SERVICE peripheral ICs connected to a μ C by polling or via interrupts. The polling method can be time-consuming, so interrupt handling is often preferable because the μ C has to take care of the peripheral only on request. However, each separate interrupt

causes the μ C to stop normal program execution, save its current state on the system stack, and vector to the interrupt's processing function. The first instructions in the interrupt function normally push some or all registers used onto the stack.

Peripherals, such as the 16550 UART, that have more than one interrupt source, may require that the μ C process more than one request at a time. Fortunately, you can write software that allows the μ C to process more than one request in one interrupt cycle. Thus, the interrupt

LISTING 1—STANDARD INTERRUPT-HANDLING SOFTWARE

```
static interrupt (0x1F) void UartIrq ( void ) /* EX7 = vector 0x1f, address 0x7c */
{
    switch ( (uiPortBase + IIR) & 6 ) /* read interrupt identification register */
    {
        case 0:
            /* do something to handle the ID 0 */
            break;
        case 2:
            /* do something to handle the ID 2 */
            break;
        case 4:
            /* do something to handle the ID 4 */
            break;
        case 6:
            /* do something to handle the ID 6 */
            break;
    }
}
```

LISTING 2—MORE EFFICIENT INTERRUPT-HANDLING SOFTWARE

```
static interrupt (0x1F) void UartIrq ( void ) /* EX7 = vector 0x1f, address 0x7c */
{
    UCHAR ucIIR;
    while ( 0 == (ucIIR = (uiPortBase + IIR) & 1 ) ) /* D0=0 --> Interrupt pending */
    {
        switch ( ucIIR & 6 ) /* Identify the interrupt source */
        {
            case 0:
                /* do something to handle the ID 0 */
                break;
            case 2:
                /* do something to handle the ID 2 */
                break;
            case 4:
                /* do something to handle the ID 4 */
                break;
            case 6:
                /* do something to handle the ID 6 */
                break;
        }
    }
}
```

overhead occurs only once. The result is improved system performance.

Listing 1 is an example of standard interrupt-handling software. Standard practice involves vectoring, pushing, and popping registers for each interrupt request. A more sophisticated function in

Listing 2 tries to handle as many interrupt requests as the peripheral requires. This function processes multiple reads to the identification register, and the process repeats until the μ C has serviced all sources. You can download both listings from EDN's Web site, www.ednmag.com.

Click on "Search Databases" and then enter the Software Center to download the file for Design Idea #2489. (DI #2489)

TO VOTE FOR THIS DESIGN,
CIRCLE No. 407

Delay line eases Spice dead-time generation

Christophe Basso, On Semiconductor, Toulouse, Cedex, France

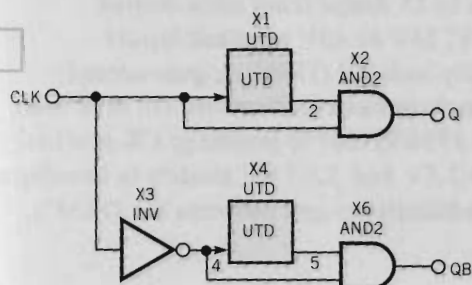
GENERATING COMPLEMENTARY clock signals in a Spice simulation is an easy task. However, this task gets much harder if you need to introduce some dead time into the signals. This difficulty is especially true when you're dealing with a variable-pulse-width-modulated switching cycle. In fact, you need to

insert a dead-time interval between the switching of any two power devices in series, such as bridge or half-bridge designs that use MOSFETs and switch-mode power supplies and that implement synchronous rectification. The dead time prevents any cross-conduction, or shoot-through, between both switches and

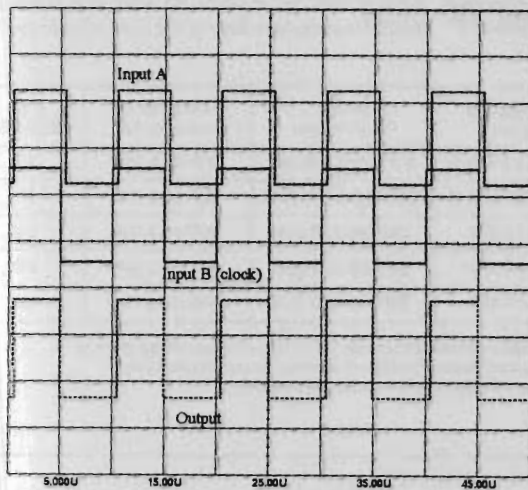
helps to reduce the associated losses. The circuit in **Figure 1a** overcomes this typical Spice problem. The input clock drives two delay lines that feature the same specifications. When the clock goes high, one input to X2's AND gate is also high. However, because of the delay line, the other input stays low for the given dead time. When both inputs are high, the output is a logic one (**Figure 1b**).

When you generate models in a proprietary syntax, the translation process to another platform is usually painful. However, thanks to common Spice3 primitives, such as the delay line, T, the trans-

Figure 1



(a)



(b)

Two AND gates and two delay lines generate a dead-time element in Spice (a). When both inputs are high, the output is a logic one (b).

LISTING 1—HALF-BRIDGE DRIVER IN ISSPICE4

Listing 1 for DI #2490

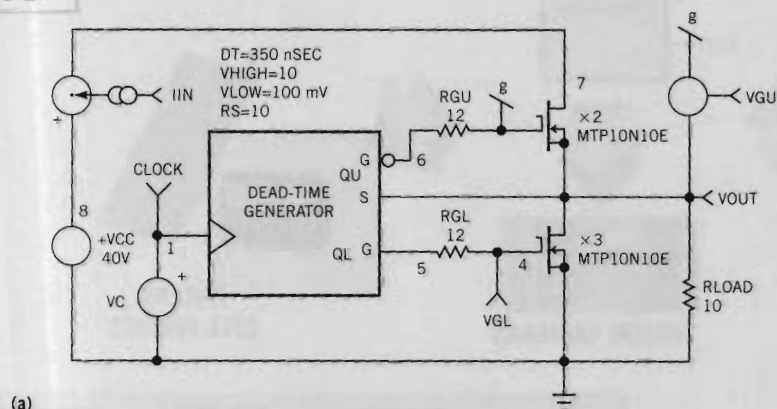
```
.SUBCKT NEWDT CLK GU SU QL (DT=500N VHIG=10V VLOW=100M RS=1
* Clock_in GateUpper SourceUpper GateLower
*
* DT: Dead time in seconds
* VHIG: Output level when high
* VLOW: Output level when low
* RS: Driver's output resistance
*
BU1 1 0 V=(V(CLK)>800M) & (V(TD1)>800M) ? {VHIG} : {VLOW}
BU2 4 SU V=V(1)
RSU 4 GU (RS)
RFLO SU 0 1G
BL 2 0 V=(V(CLKB)>800M) & (V(TD2)>800M) ? {VHIG} : {VLOW}
RSL 2 QL (RS)
X1 CLK TD1 UTD PARAMS: TD=DT
X2 CLKB TD2 UTD PARAMS: TD=DT
X3 CLK CLKB INV
.ENDS
*INCLUDE DEAD.LIB
*****
.SUBCKT UTD 1 2 {TD=???}
*
*Parameters K=GAIN TD=DELAY
RIN 1 0 1E15
E1 3 0 1 0 1
T1 3 0 2 0 ZO=1 TD={TD}
R1 2 0 1
.ENDS
**** 1 INPUT INVERTER ****
.SUBCKT INV 1 2
B1 4 0 V=V(1)>800M ? 0 : 5V
RD 4 2 100
CD 2 0 10P
.ENDS INV
```

LISTING 2—HALF-BRIDGE DRIVER IN PSPICE

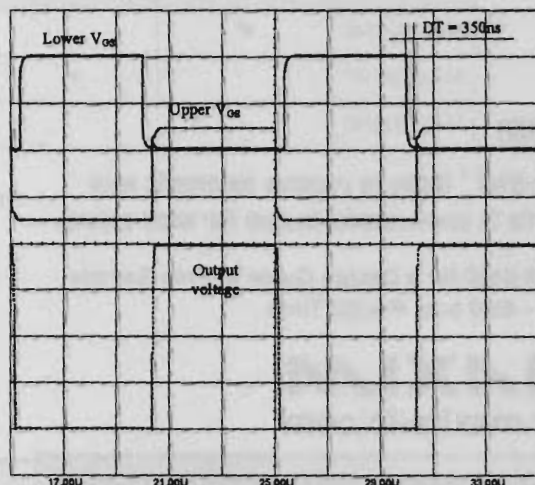
```
.SUBCKT NEWDT CLK GU SU QL PARAMS: DT=500N VHIG=10V VLOW=100mV
RS=10
* Clock_In GateUpper SourceUpper GateLower
*
* DT: Dead time in seconds
* VHIG: Output level when high
* VLOW: Output level when low
* RS: Driver's output resistance
*
EBU1 1 0 VALUE = { IF ( (V(CLK)>800M) & (V(TD1)>800M), {VHIG}, {VLOW} ) }
EBU2 4 SU VALUE = { V(1) }
RSU 4 GU {RS}
RFLO SU 0 1G
EBL 2 0 VALUE = { IF ( (V(CLKB)>800M) & (V(TD2)>800M), {VHIG}, {VLOW} ) }
RSL 2 QL {RS}
X1 CLK TD1 DL PARAMS: TD={DT}
```

```
X2 CLKB TD2 DL PARAMS: TD={DT}
X3 CLK CLKB INV
.ENDS
*****
.SUBCKT DL 1 2 PARAMS: TD=500n
*
RIN 1 0 1E15
E1 3 0 1 0 1
T1 3 0 2 0 ZO=1 TD={TD}
R1 2 0 1
.ENDS DL
**** 1 INPUT INVERTER ****
.SUBCKT INV 1 2
EB1 4 0 VALUE = { IF ( V(1)>800M, 0, 5V ) }
RD 4 2 100
CD 2 0 10P
.ENDS INV
```

Figure 2



(a)



(b)

lation of this generator is easy to implement. The netlists in listings 1 and 2 implement a half-bridge driver with a floating upper output in IsSpice4 (Intusoft) and Pspice (OrCAD), respectively. The BL (Listing 1) and EBL (Listing 2) inline equations implement the AND gates of Figure 1a, which saves you from using a subcircuit arrangement. Typical applications include half-bridge drivers and synchronous rectifiers. You can easily tailor any output polarity by reversing the corresponding Spice element. For instance, if you want to reverse the upper generator, BU1, in Listing 1, simply replace the line $V=(V(CLK)>800M) \& (V(TD1)>800M) ? \{VHIG\} : \{VLOW\}$ with $V=(V(CLK)>800M) \& (V(TD1)>800M) ? \{VLOW\} : \{VHIG\}$.

Figure 2a portrays a typical application of the dead-time generator in a simplified half-bridge driver, and Figure 2b shows the corresponding IsSpice4 waveforms. You can download both listings from EDN's Web site, www.ednmag.com. Click on "Search Databases" and then enter the Software Center to download the file for Design Idea #2490. (DI #2490)

A typical application for the Spice dead-time generator is for simulating the operation of a half-bridge driver (a). IsSpice4 waveforms show a dead time of 350 nsec (b).

TO VOTE FOR THIS DESIGN,
CIRCLE NO. 408

Comparison macro for PIC processors

Dennis M Nagel, Dennis M Nagel Inc, Delray Beach, FL

IF YOU EVER get tired of trying to remember the subtleties of the "carry" status bit every time you want to use the subtract instruction to perform a comparison, the macro in **Listing 1** can help. (You can download a copy of the listing from EDN's Web site, www.ednmag.com. Click on "Search Databases" and then enter the Software Center to download the file for Design Idea #2493.) The macro contains all of the nuances, once and forever. The macro reads like a sentence: branch to target if ram-register is [comparison condition] a literal value. The comparison conditions available are "equal-to," "not-equal-to," "below," and "above-or-equal." The words "below," "above," and others adhere to the Intel/Microsoft assembly-language convention of referring to "unsigned" comparisons for which the byte value can range only from 0x00 (decimal zero) to 0xFF (decimal 255).

Although you can use this macro for "equal" and "not-equal" comparisons, its real power comes in examining a value in

LISTING 1—COMPARISON MACRO

```
#define eq      1      ; if equal
#define ne      2      ; if not equal
#define bl      3      ; if below (unsigned comparison)
#define ae      4      ; if above or equal (unsigned)
#define WREG    0xFFFD ; if immediate-value specified as the W reg

b2          MACRO      target, ram_reg, cond, lval
nolist
if lval != WREG      ; if lval spec'd as "WREG", don't move lval to W
    movlw lval
endif
; subtract will set carry if ram_reg >= lval, clear it if reg < lval (unsigned)
subwf ram_reg, W      ; ram_reg minus W -> W, ram_reg unchanged
if cond == eq          ; if cond is "eq", skip if non 0, branch if 0
    skpnz
else
    if cond == ne      ; if cond is "ne", skip if 0, branch if non 0
        skpz
    else
        if cond == bl  ; if cond is "bl", skip if carry, branch if not
            skpc
        else
            if cond == ae ; if cond is "ae", skip if not carry, else branch
                skpnc
            else
                error "b2 macro condition argument not a valid choice"
            ; cond = ae
            ; cond = bl
            ; cond = ne
            ; cond = eq
        b target ; skip this if condition not true, take it if true
    list
ENDM ; b2
```

a range or in a window of values. As an example of the macro's use, suppose you want to execute some code, but only if some ram-register is 0x6C or more but not greater than 0x93 (0x94 to 0xFF). You would use:

```
b2 continue_label1, ram-register, b1,
    0x6C
```

b2 continue_label1, ram-register, ae, 0x94

source code here to execute only if ram-register = 0x6C to 0x94 inclusive.

continue_label1:

A variation of usage is available if you want to perform a comparison between a ram-register and the "W" register instead of a literal value. An example of this usage is:

```
b2 continue_label2, ram-register, b2, WREG
```

source code here to execute only if ram-register is below (less than and not equal to) the current value in the W register.

continue_label2:

Note that these examples destroy the value in the "W" register. (DI #2493)

To VOTE FOR THIS DESIGN,
CIRCLE NO. 409

Bridging enhances filter close-in selectivity

Richard M Kurzrok, RMK Consultants, Queens Village, NY

GENERAL FILTERS are bandpass filters that usually employ bridging couplings between nonadjacent interstage couplings (**Reference 1**). This class

of filters also includes bridging coupling across the filter input and output ports. The implementation of input-to-output bridging already exists for a one-pole fil-

ter (**Reference 2**). For a two-pole filter, dielectric resonators help achieve input-to-output bridging coupling (**Reference 3**).

You can also implement a two-pole

LC-bandpass filter using input-to-output bridging (Figure 1). This general filter provides enhanced close-in selectivity by adding a single bridging inductor across the input and output of a conventional bandpass filter. This relatively obscure passive filter may be useful in some applications, and you can probably realize similar filter performance using an active circuit.

You can compare the performance of this general filter to the performance of a conventional two-pole bandpass filter that has a convenient center frequency of 20 MHz. The design parameters are for a lossless 0.01-dB Chebyshev response with a 3-dB bandwidth of 2 MHz and input and output impedances of 50Ω. Table 1 shows the relative measured amplitude response data. Center-frequency insertion loss was 0.8 dB, corresponding to inductor unloaded Q's of about 150. The measured relative 3-dB bandwidth was 2.1 MHz. The filter response is asymmetrical due to the frequency sensitivity of the capacitive input, output, and interstage couplings.

The general filter adds a bridging inductor from the input to the output of the conventional bandpass filter (Figure 1). The measured center-frequency insertion loss was 0.7 dB, and Table 2 shows the relative amplitude-response data. The

TABLE 1—RESPONSE OF A TWO-POLE CONVENTIONAL BANDPASS FILTER

Frequency (MHz)	Insertion loss (dB)
15	40.3
18	17.8
19	6.2
19.2	4.0
19.4	2.0
19.6	0.8
19.8	0.2
20	0
20.2	0
20.4	0
20.6	0.2
20.8	0.7
21	1.7
21.2	3.0
21.4	4.4
22	8.6
23	14.2
24	17.8
30	26.2

general two-pole filter provides nearby rejection peaks with degraded far-out selectivity. The filter exhibits this type of behavior for upper and lower stopbands. You adjust the variable capacitors with the enclosure cover removed. As filter bandwidth becomes smaller, alignment-tool access holes in the cover become necessary. (DI #2491)

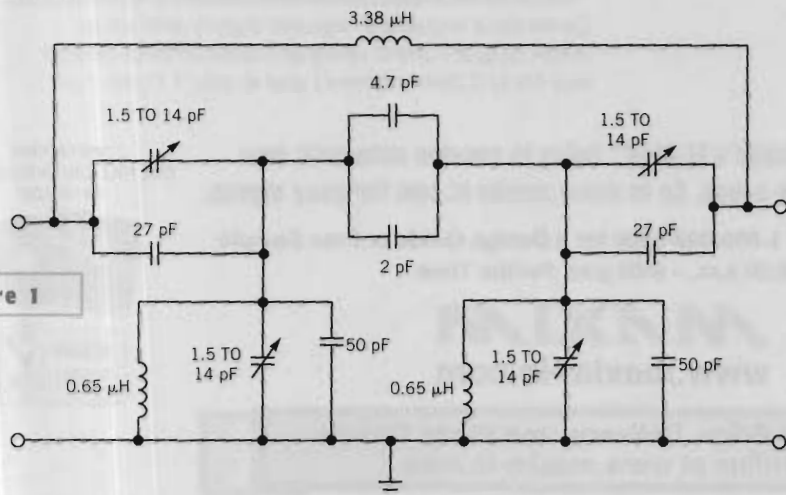
TABLE 2—RESPONSE OF A TWO-POLE GENERAL FILTER

Frequency (MHz)	Insertion loss (dB)
13	9.8
15	11.2
17	14
18	22
18.2	29.2
18.3	>34 (peak)
18.4	28.6
18.6	18.2
18.8	12.6
19	7.3
19.2	4.0
19.4	1.6
19.6	0.5
19.8	0.2
20	0
20.2	0
20.4	0.2
20.6	0.8
20.8	1.8
21	3.6
21.2	6.6
21.4	8.0
21.6	10.4
21.8	13.1
22	16.2
22.5	24.6
23	>39 (peak)
23.5	29.8
24	25.2
25	22
30	20.4

REFERENCES

1. Kurzrok, RM, "General three-resonator filters," *EDN*, May 1966, pg 92.
2. Kurzrok, RM, "Single component changes bandpass into general filter," *Electronics*, April 18, 1966 pg 95.
3. Cohn, SB, "Microwave filters containing high-dielectric resonators," presented at Clearwater, FL, May 5 to 7, 1965, and printed in *G-MTT Digest*, pg 49.

Figure 1



A bridging inductor across the input and output of a conventional two-pole bandpass filter creates a general filter that provides nearby rejection peaks with degraded far-out selectivity.

TO VOTE FOR THIS DESIGN,
CIRCLE NO. 410

Notch filter is insensitive to component tolerances

John Guy, Maxim Integrated Products, Sunnyvale, CA

MANY APPROACHES FOR creating notch filters, which reject a narrow band of frequencies and pass all others, are unsatisfactory because they allow the component tolerances to interact. The circuit in **Figure 1a** overcomes this limitation and enables easy calculation of the component values for a desired notch frequency.

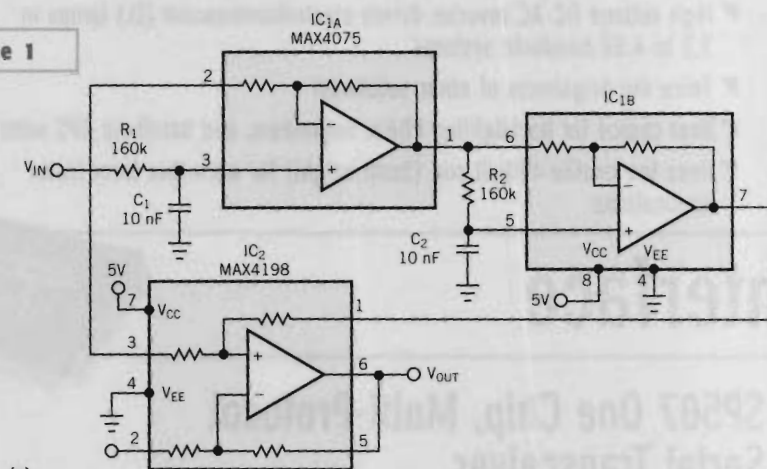
Two allpass filter stages, IC_{1A} and IC_{1B} , create a dc-accurate, 180° phase shift at the cutoff frequency. Each op amp in IC_1 includes gain resistors that match to within 0.1%. This tight tolerance eliminates the need for trimming in most applications. Summing this phase-shifted signal with the input produces a cancellation that produces the notch.

At low frequencies for which the impedance of C_2 is negligible, the circuit forms a voltage follower and produces no phase inversion. For high frequencies, however, this capacitor acts as a short circuit that causes the amplifier to act as a unity-gain inverter with the associated 180° phase shift. Phase behavior for the resulting allpass filter is identical to that of a single RC pole and produces 90° of phase shift at the resonant frequency, which is equal to $1/2\pi R_1 C_1$ and $1/2\pi R_2 C_2$.

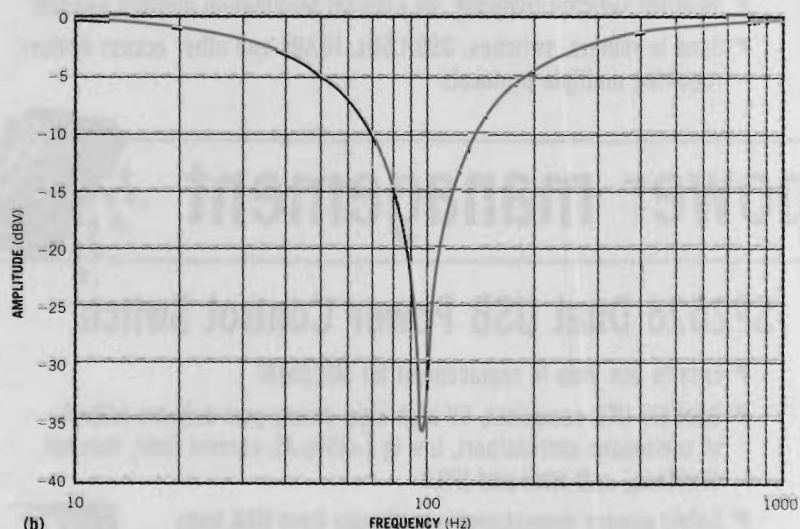
R_1 , R_2 , C_1 , and C_2 affect only the notch frequency and not its depth. Conversely, the integrated resistors in IC_1 affect only the depth of the notch and not its frequency. If you require a highly accurate notch frequency, specify R_1 , R_2 , C_1 , and C_2 accordingly or simply trim one of the two resistors. IC_2 is a precision differential amplifier that the circuit uses as a matched summing amplifier. Note that the inverting input is left unconnected.

Figure 1b shows the circuit's performance with 5% resistors and 20% capacitors, all unmatched. To produce a deeper notch, you can trim the circuit by adding a 100Ω resistor in series with Pin 3 of IC_2 . You can also add a 200Ω poten-

Figure 1



(a)



(b)

Summing V_{IN} with the output of IC_1 's phase-shifting allpass filter results in a notch response (a). Operating the circuit with 5% values for R_1 and R_2 and 10% values for C_1 and C_2 produces a notch frequency of approximately 99 Hz (b).

tiometer in series with Pin 1 of IC_2 and adjust this potentiometer for maximum rejection at the desired frequency. (DI #2492)

TO VOTE FOR THIS DESIGN,
CIRCLE No. 411